Organizati

P.O. BOX 1450
ALEXANDRIA, VA 22313-1450
IF UNDELIVERABLE RETURN IN TEN DAYS

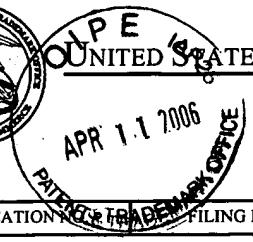
OFFICIAL BUSINESS

UNITED STATES POSTAGE
U.S. OFFICIAL MAIL
PENALTY FOR
PRIVATE USE $300
PITNEY BOWES
02 1A
0004204479
$ 01.59⁰
MAR 31 2006
MAILED FROM ZIP CODE 22314

AN EQUAL OPPORTUNITY EMPLOYER

2145

UNITED STATES PATENT AND TRADEMARK OFFICE



UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/934,738	08/22/2001	Ingo Molnar	019322-000340	9016

24239 7590 03/31/2006

MOORE & VAN ALLEN PLLC
P.O. BOX 13706
Research Triangle Park, NC 27709

EXAMINER	
CHOUDHURY, AZIZUL Q	
ART UNIT	**PAPER NUMBER**
2145	

DATE MAILED: 03/31/2006

Please find below and/or attached an Office communication concerning this application or proceeding.

Office Action Summary	Application No.	Applicant(s)	
	09/934,738	MOLNAR, INGO	
	Examiner	Art Unit	
	Azizul Choudhury	2145	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on *09 January 2006*.

2a)☐ This action is **FINAL**. 2b)☒ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1-14* is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1-14* is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.

10)☒ The drawing(s) filed on *22 August 2001* is/are: a)☒ accepted or b)☐ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1) ☒ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☐ Information Disclosure Statement(s) (PTO-1449 or PTO/SB/08) Paper No(s)/Mail Date _____.
4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____.
5) ☐ Notice of Informal Patent Application (PTO-152)
6) ☐ Other: _____.

Detailed Action

This office action is in response to the correspondence received on 1/9/06.

Claim Rejections - 35 USC § 101

35 U.S.C. 101 reads as follows:

> Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title.

Claims 5-8 are rejected under 35 U.S.C. 101 because the claimed invention is directed to non-statutory subject matter. The claims describe a software product and software is not patentable.

- Claims 5-8 disclose a computer program product, which is a non-tangible product. The examiner recommends amending the claims to claim a tangible product such as a computer-readable medium, which stores a computer program product. However the examiner also reminds the applicant's representatives that the all claimed features must be supported by the design specifications.

Claim Rejections - 35 USC § 112

The following is a quotation of the second paragraph of 35 U.S.C. 112:

> The specification shall conclude with one or more claims particularly pointing out and distinctly claiming the subject matter which the applicant regards as his invention.

Claims 1-2, 5-6, 9-11 and 13-14 are rejected under 35 U.S.C. 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention. It is unclear what an object is

since a plurality of objects exist within the art. The specifications fail to disclose an appropriate definition defining an object.

Claim Rejections - 35 USC § 102

The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (e) the invention was described in (1) an application for patent, published under section 122(b), by another filed in the United States before the invention by the applicant for patent or (2) a patent granted on an application for patent by another filed in the United States before the invention by the applicant for patent, except that an international application filed under the treaty defined in section 351(a) shall have the effects for purposes of this subsection of an application filed in the United States only if the international application designated the United States and was published under Article 21(2) of such treaty in the English language.

Claims 1-14 are rejected under 35 U.S.C. 102(e) as being anticipated by Challenger et al (US Pat No: 6,256,712), hereafter referred to as Challenger.

1. With regards to claims 1, 5, 9 and 11, Challenger teaches in a communication server, a method of responding to a client application, the method comprising the steps of: a cache disposed in an operating system kernel; receiving from the client application an application protocol request corresponding to a response that can be displayed as a combination of a dynamic protocol object and a static protocol object; creating at the server the dynamic protocol object; sending the dynamic protocol object to the client application; retrieving the static protocol object from a cache disposed in an operating system kernel; and sending the static protocol object to the client application (Challenger discloses a design

enabling the updating content within a server so that updated content is submitted to the client. The design allows for current copies of both dynamic and static data (objects) to be cached within the server (column 2, lines 5-8, Challenger). The cached data (objects) is consistently updated (column 2, lines 54-55, Challenger). When required, the data (objects) are dynamically rebuild the objects and provide the client with updated content (column 2, line 53 – column 3, line 34, Challenger). Finally, the use of a cache/buffer/registry within an operating system of a computer is inherent).

2. With regards to claims 2, 6, 10, 13 and 14, Challenger teaches the method wherein the cache disposed within the operating system kernel is a protocol object cache (Challenger's design allows for caches (column 2, lines 5-8, Challenger) (column 5, lines 51-52, Challenger)).

3. With regards to claims 3, 4, 7, 8 and 12, Challenger teaches the method wherein the application protocol request and the reply are formatted according to a hypertext transmission protocol (HTTP) (Challenger's design allows for HTTPD (Figure 30A, Challenger). Hence, HTTP is supported).

Remarks

The amendment received on January 9, 2006 has been carefully examined but is not deemed fully persuasive. In lieu of the claim amendments, the claim objections

have been withdrawn. With regards to the applicant's representative's remarks, two primary points of contention are addressed. As for the first point of contention, the applicant's representative remarks on how the term "object" is well known in the art. However, the examiner would like to point out how the term "object" lacks a clear definition within the specifications. "Object" is a broad and indefinite term and is open to a variety of interpretations. Hence, the 112-type rejection continues to stand. As for the second point of contention, the applicant's representative states that the claimed design features a response that can be displayed as a combination of a dynamic protocol object and a static protocol object and that the prior art teaches no such traits. First, the examiner has interpreted the claimed "objects" to be equivalent to data. Then, the examiner referred to the Challenger art and stated: "The design allows for current copies of both dynamic and static data (objects) to be cached within the server (column 2, lines 5-8, Challenger). The cached data (objects) is consistently updated (column 2, lines 54-55, Challenger). When required the data (objects) dynamically rebuilds the objects and provides the client with updated content (column 2, line 53 – column 3, line 34, Challenger)." In addition, the examiner states that the use of a cache/buffer/registry within an operating system of a computer is inherent.

Conclusion

Any inquiry concerning this communication or earlier communications from the examiner should be directed to Azizul Choudhury whose telephone number is (571) 272-3909. The examiner can normally be reached on M-F.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Jason Cardone can be reached on (571) 272-3933. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

AC

JASON CARDONE
SUPERVISORY PATENT EXAMINER

Notice of References Cited	Application/Control No. 09/934,738	Applicant(s)/Patent Under Reexamination MOLNAR, INGO	
	Examiner Azizul Choudhury	Art Unit 2145	Page 1 of 1

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
*	A	US-6,256,712	07-2001	Challenger et al.	711/141
*	B	US-6,591,266	07-2003	Li et al.	707/10
	C	US-			
	D	US-			
	E	US-			
	F	US-			
	G	US-			
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	Huang, Yixiu et al, "Divergence Caching in Client-Server Architectures," Sept. 1994, IEEE, pp. 131-139.
	V	Porto, Fabio A. M. et al. "Persistent Object Synchronization with Active Relational Databased," Aug. 1999, IEEE, pp. 53-62
	W	
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

Divergence Caching in Client-Server Architectures*

Yixiu Huang Robert H. Sloan Ouri Wolfson
Electrical Engineering and Computer Science Dept.
University of Illinois at Chicago
Chicago, IL 60607

Abstract

In this paper we propose a new mechanism, divergence caching, for reducing access and communication charges in accessing on-line database servers. The objective is achieved by allowing tolerant read requests, namely requests that can be satisfied by out-of-date data. We propose two algorithms based on divergence caching, Static and Dynamic. The first is appropriate when the access pattern to an object in the database is fixed and known, and the latter is appropriate in other cases. We analyze these algorithms in the worst case and the expected case.

1 Introduction and overview

Users will soon have on-line access to a large number of databases, often via wireless networks. The potential market for this activity is estimated to be billions of dollars annually, in access and communication charges. For example, passengers will access airline and other carriers' schedules, and weather information. Investors will access prices of financial instruments, salespeople will access inventory data, callers will access location-dependent data (e.g. where is the nearest taxicab, see [3, 9]) and route-planning computers in cars will access traffic information.

Because of limited bandwidth, wireless communication is more expensive than wire communication. For example, a cellular telephone call costs about $0.35 per minute, and RAM Mobile Data Corporation charges on average $0.08 per data message to or from a mobile computer (the actual charge depends on the length of the message).

Additionally, database publishers will also charge an access fee for each transmission of an object or data item to a client. Thus each such transmission will incur both access and communication charges. Similarly, today when calling a 900 telephone number the caller is charged two separate fees, one for communication and the other for access.

*This research was supported in part by NSF grant IRI-9224605 and AFOSR grant F49620-93-1-0059.

It is clear that for users who perform hundreds of accesses each day, access and communication charges can become very expensive. Therefore, it is important that client computers access on-line database servers in a way that minimizes these charges.

In this paper, we explore the minimization of these charges via a new mechanism called *divergence caching*. Its objective is to reduce the number of transmissions of an object from an on-line database to a client computer. It does so by using the following two techniques for each object in the on-line database.

- The first technique is using *tolerant reads*. A Client Computer (CC) issues reads (and possibly writes) for a data object. In order to reduce access and communication charges, each read is associated with a natural number representing the divergence-tolerance for the read. For example, read(IBM,3) represents a request to read IBM's stock price (i.e. the object) with a tolerance of 3. This read can be satisfied by any of the three latest versions of IBM's stock price; in other words, it can be satisfied by a version that is up to two updates behind the most recent version.

- The second technique is *automatic refresh*. Its purpose is to eliminate the need for object transmission for every read, and it does so as follows. The Server Computer (SC) that stores the on-line database receives all the updates of the object. We are not concerned with the source of these updates. For every client computer that reads the object, the SC has a refresh rate r. This means that the version of the object cached at the CC is at most $r-1$ updates behind the version at the SC. To achieve this, the SC automatically propagates to the CC the r'th version since the last transmission of the object to the CC. The CC saves the last version of the object it received from the SC. Thus, those reads at the client computer with a tolerance greater than r can be satisfied locally; i.e., without access to the on-line database (which avoids the access and communication charges). Therefore, access and communication

0-8186-6400-2/94 $4.00

charges are incurred only for automatic refresh, and for each read with a tolerance that is lower than r.

The refresh rate can have any value between 1 and infinity. A refresh rate of 1 means that the CC has a regular copy of the object (i.e. the object is replicated at the CC in the traditional sense), and each update of the object is propagated to the CC. A refresh rate of infinity means that the CC does not have a copy of the object; each read, regardless of its tolerance, will require a transmission from the SC to the CC (even when the object has not changed since the last read). The optimal refresh rate, i.e. the refresh rate that minimizes the object transmissions, depends on the ratio between the frequency of updates at the SC on one hand, and the frequency and tolerance of reads at the CC on the other hand.

Note that this paradigm is only appropriate for cases where the client does not need the most recent value of the object. We envision these methods being used, for instance, by a relatively passive investor to monitor her portfolio, or by a basketball fan (perhaps at work) to get scores during an ongoing game, or by a customer monitoring the general level of an inventory item (e.g. the number of compact cars available at a particular rental location, with each update representing a rental or a return).

In this paper we propose and analyze two related algorithms, Static Divergence Caching (SDC) and Dynamic Divergence Caching (DDC). The SDC algorithm works as described above, and it has a fixed refresh rate. Observe that actually there is an infinite number of SDC algorithms, one for each refresh rate. One of the problems that we solve in the paper is to determine the optimal refresh rate for a given frequency of writes and reads of each tolerance. Specifically, we assume a Poisson distribution for the writes, a Poisson distribution for reads of each tolerance, and we assume that we know the intensities (i.e. λ parameters) of each type of request; and we solve the problem of determining the refresh rate that will minimize the expected cost of a request.

The DDC algorithm is similar to the static one, except that the refresh rate varies over time. It does so since we assume that the intensities of the Poisson distributions are unknown or they vary over time. Our DDC algorithm learns these by "watching" a sliding window of read-write requests, and based on it the algorithm continuously adapts the refresh rate to the current intensities. The DDC algorithm also has an infinite number of variants, one for each window-size. The DDC algorithm is distributed, and it varies the refresh rate between 0 and infinity. It is implemented by software residing on both the client and the server computers.

In addition to the expected case, we also analyze the worst case for both algorithms, and we show that in the worst case the DDC algorithm is superior to the SDC algorithm.

Finally, we analyzed the DDC algorithm experimentally. We have determined that the appropriate window size is approximately 23. We have also determined that if the distribution parameters are fixed, then the DDC algorithm comes within 15–45% of the optimal SDC algorithm, i.e. the static algorithm with the optimal refresh rate for the given distribution parameters. For fixed distribution parameters the DDC algorithm is used when these parameters are unknown a priori. If the distribution parameters vary over time, then the DDC algorithm with a window size of 23 is strictly superior to any static algorithm.

The rest of the paper is organized as follows. In the next section, we give a precise definition of our model. In Section 3, we make a detailed mathematical analysis of the Static Divergence Caching algorithm; particularly, we determine the optimal refresh rate for the case where the probability distribution of the random process generating the read and write requests is known and fixed over time. Section 4 presents the Dynamic Divergence Caching algorithm, which is appropriate when the probability distribution of the random process changes over time or is simply unknown. Section 5 gives a theoretical analysis of that algorithm's performance. In Section 6 we give experimental results obtained by simulating our techniques on various data sets. We compare this paper to the relevant literature in Section 7, and we make a few concluding remarks in Section 8.

2 Model of Problem

Our client-server system consists of a server computer, SC, and a client computer, CC. We consider a single data object that is always updated and stored at the SC, and that the CC requests a copy of it from time to time. In this paper we concentrate on the case where the CC may not need the absolutely most recent version of the data object.

A *request* is either a write, denoted w, issued by the SC or a read issued by the CC.[1] Each write request creates a new version of the object. Each read request has a *tolerance* t, specifying how recent a version of the object is required. We denote such a read by $r(t)$. We assume that the read tolerance is an integer in the range $1, \ldots, M$. A read tolerance of 1 indicates that the most recent version of

[1] In general, there may also be writes issued by the CC and reads issued by the SC. However, since the SC always has the latest copy of the object, those requests always have the same fixed cost and therefore we ignore them.

the object is requested; a read tolerance of k indicates that any of the last k versions will do.

A *schedule* is a finite sequence of requests; for example, $w, w, r(1), w, r(3), r(2), w$. For the purpose of analysis, we assume that the requests are sequential. The reason for this is that the reads are all generated by the CC, so they are sequential. Although a read and a write request or two write requests could be generated concurrently, at some point a concurrency control mechanism will serialize them, so our analysis still holds.

Throughout, we assume that requests are generated by Poisson processes. In particular, we assume that there are M separate Poisson processes, each generating read requests with a tolerance $1 \leq i \leq M$; let process i have intensity λ_{r_i}. Thus λ_{r_i} is the average number of read requests with tolerance i per time unit. Another Poisson process generates writes at the SC; call its intensity λ_w.

Another possible model of our problem would involve a single Poisson process generating read requests, with the tolerance of a read being determined by picking from the set of tolerances $\{1, 2, \ldots, M\}$ according to some fixed probability distribution. This model, however, is equivalent the model we specified above, since the sum or combination of M Poisson processes is itself a Poisson process.

One of the problems studied in this paper is, for the Static Divergence Caching algorithm, how often the CC should have the object regularly delivered. We will call this parameter the *refresh rate*. A refresh rate of k means that the object is automatically transmitted to the CC every time the object has been updated by k writes without having been sent to the CC in the meantime. Thus, when the refresh rate is k, the CC always has one of the k most recent versions of the object in its local memory, and can satisfy any read request with a tolerance of k or more with that local version. The CC may choose never to have the object regularly delivered; this corresponds to a refresh rate of infinity.

Observe that a refresh rate of k does not mean that exactly one in every k writes is propagated to the CC. If the CC solicits a refresh (to satisfy a read with a low tolerance) after $k/2$ writes, then that refresh will reinitialize the refresh counter. In other words, an automatic, or solicited refresh will occur after k writes only if in the meantime there was no solicited refresh. Observe that performing an unsolicited refresh exactly once for every k writes would strictly increase the number of object transfers compared to our proposed scheme.

We assume that each message containing the data object costs 1, and each read request, i.e. a control message, costs ω. Intuitively, $0 \leq \omega < 1$, but we will not assume this unless we explicitly say so. In our model, reads with tolerance less than the refresh rate cost $1 + \omega$, because the CC must send a control message of cost ω to "special order" the data object, and then pay 1 for the transmission of the data object. Reads with tolerance greater than or equal to the refresh rate have zero cost. A write costs 1 if it is propagated to the CC, otherwise it costs 0.

Note that the case $\omega = 0$ is of particular interest. It models the situation in mobile computing where communication is by cellular telephone calls, where there is a charge for the first minute or part thereof. If we assume that a remote read request and the response are executed within one minute, then each remote read or propagated write costs 1.

3 Fixed and known distributions

In this section we assume that the λs and ω are fixed and known a priori, and we are using the Static Divergence Caching algorithm. We develop the expected cost per time unit as a function of the refresh rate, and we show how to find the minimum of that function. This minimum is the optimal refresh rate.

Our first goal is to compute the expected cost per time unit for any given fixed refresh rate k. (The time unit is the period of time for the intensities λ; i.e. λ_w is the expected number of writes during one time unit). The case $k = \infty$ is straightforward. We pay $1+\omega$ for every read, and nothing for any of the writes. Thus the cost per unit time is $(1 + \omega) \sum_{t=1}^{M} \lambda_{r_t}$. The case $k = 1$ is also straightforward. We pay 1 for each write and nothing for any reads, so the expected cost per unit time is λ_w.

Otherwise, for fixed integer $1 < k < \infty$, the requests we might possibly pay for are writes, and reads with tolerance less than k. We call such requests *relevant* and reads with tolerance at least k *irrelevant*. Notice that irrelevant requests are always free. Notice also that, as explained in the previous section, the number of writes we pay for is not $1/k$'th of all the writes (which would make the derivation of the optimal refresh rate easier).

Put $r(k) = \sum_{t=1}^{k-1} \lambda_{r_t}$. This is the intensity of the Poisson process generating all reads that we will have to pay for; once we have fixed k the individual values of the λ_{r_t} for $t < k$ no longer matter.

Put $\theta_k = \lambda_w/(\lambda_w + r(k))$, and put $\theta = \theta_{M+1}$. (Notice that $\theta_1 = 1$.) Thus θ is the probability that an arbitrary request is a write, and the probability that a relevant request is a write is θ_k.

Now from here on in, we condition all probabilities and expectations on the event that the request being considered is relevant. (At the end, we will need to multiply through by the probability of this event, which is $(\lambda_w + r(k))/(\lambda_w + r(k) + \sum_{i=t}^{M} \lambda_{r_i})$.)

The probability that we pay for an arbitrary request is

the sum of the probabilities that the request is a relevant read, which is clearly $1-\theta_k$, and that the request is a write we pay for. Hence we need to calculate the probability that a relevant request is a write we pay for. In fact, we pay for a write in a schedule whenever the sequence of relevant requests leading up to and including that request is either a read followed by k writes, or a read followed by $2k$ writes, or a read followed by $3k$ writes, or.... These events are all disjoint, so the probability of any of them occurring is just the sum of the probability of each. This sum in the limit is

$$\begin{aligned}\Pr[\text{Request is write we pay for}] &= \sum_{n=1}^{\infty}(1-\theta_k)(\theta_k)^{nk} \\ &= \frac{(1-\theta_k)\theta_k^k}{1-\theta_k^k}.\end{aligned}$$

Notice that the precise probability depends on how long the communications have been taking place. However, the geometric sum converges very quickly, so the limiting value calculated above should be a very close approximation after the first few requests.

Thus the expected cost of one arbitrary relevant request is

$$E(\text{cost}) = (1-\theta_k)\left(1+\omega+\frac{\theta_k^k}{1-\theta_k^k}\right). \quad (1)$$

Recall that this was conditioned on the request being relevant, so the actual expected cost of one arbitrary request is

$$\frac{(r(k)+\lambda_w)}{\lambda_w+r(k)+\sum_{i=k}^{M}\lambda_{r_i}}(1-\theta_k)\left(\omega+\frac{1}{1-\theta_k^k}\right) = r(k)\left(\omega+\frac{1}{1-\theta_k^k}\right)\Big/\left(\lambda_w+r(k)+\sum_{i=k}^{M}\lambda_{r_i}\right) \quad (2)$$

since, by the definition of θ_k, $(r(k)+\lambda_w)(1-\theta_k) = r(k)$.

The expected cost per unit time is the quantity specified by Equation 2 times the expected number of requests per unit time, which is just the denominator of that fraction. Thus for a refresh rate of $1 < k < \infty$ the expected cost per unit time is

$$\text{E(cost)} = r(k)\left(\omega+\frac{1}{1-\theta_k^k}\right). \quad (3)$$

Intuitively, Equation 3 reflects the balancing of two opposite influences on the cost. As k increases, $r(k)$ increases. That is, we pay more for "special orders" if we have a higher refresh rate. On the other hand, as k increases, the factor $1/(1-\theta_k^k)$ decreases, since for a fixed value of θ_k it would decrease as the exponent k increases, and in addition, θ_k decreases with k. This factor corresponds to the fact that as k increases the amount we pay for writes at the SC decreases.

Putting Equation 3 together with the extreme values for the refresh rate we get that the expected cost per unit time for a fixed refresh rate k is

$$\text{E(cost)} = \begin{cases} \lambda_w & \text{for } k=1 \\ \left(\omega+\frac{1}{1-\theta_k^k}\right)\sum_{j=1}^{k-1}\lambda_{r_j} & \text{for } 1<k<\infty \\ (\omega+1)\sum_{j=1}^{M}\lambda_{r_j} & \text{for } k=\infty \end{cases} \quad (4)$$

Theorem 1 *In the Static Divergence Caching algorithm the minimum cost per unit time can never be achieved for any finite refresh rate greater than M.*

Proof Straightforward based on the cost function in Equation 4 that for any $k > M$ the second line is bigger than the third line. □

Thus, assuming that all the λ's are fixed and known, the algorithm for finding the optimal refresh rate is trivial. All one must do is compute the $M+1$ different costs associated with the refresh rates of $1, 2, \ldots, M$ and ∞, according to Equation 4, and choose the minimum cost refresh rate.

4 The dynamic divergence caching algorithm

The Dynamic Divergence Caching algorithm works for distribution parameters (λ's) that are unknown and that may vary over time. The algorithm varies the refresh rate of an object x at the CC. It does so by computing the λ's based on a window of the k latest relevant read and write requests, using formula 4 to recompute the optimal refresh rate, and establishing it as the new refresh rate. The new refresh rate may be different than the previous one since the λ's change in a sliding window.

Now we explain the algorithm in detail. Recall, the relevant reads are issued at the CC, and the relevant writes are issued at the SC. At any point in time there is a refresh rate, r. Each read at the CC with a tolerance higher than r is satisfied locally, and each read at the CC with a tolerance lower than r results in a refresh request to be sent to the SC; the SC responds by refreshing x, i.e., sending the latest version of x. The SC also performs an unsolicited refresh of x when it receives r consecutive write requests since the last refresh of x (this last refresh may be either solicited or unsolicited).

Adaptation of the refresh rate occurs at each refresh point (solicited or not), as follows. At every point in time, the SC maintains the *write-sliding-window*, i.e. the set of times of the last k write requests. Each time a new write is received at the SC its time stamp is added to the write-sliding-window, and the smallest time stamp in the window is deleted. At every point in time, the CC maintains the

read-sliding-window, i.e. the time and tolerance of the k latest reads. Specifically, the CC maintains a set of k pairs (i, t) where i is a tolerance and t is a time stamp.

When the CC initiates a refresh request, the SC computes the new refresh rate as follows. The CC piggybacks the read-sliding-window on each refresh request. Before refreshing x, the SC uses the times in the read-sliding-window and the write-sliding window in order to compute the *request-numbers-window*; it is the number of writes, the number of reads with tolerance 0, the number of reads with tolerance 1, the number of reads with tolerance 2, etc. for the last k read-write requests. These numbers are taken as the λ_w and λ_r's. Then the SC uses Formula 4, in order to compute the optimal refresh rate; this will become the new refresh rate. Then the SC responds to the refresh request, and it informs the CC of the new refresh rate by piggybacking the rate on the refresh value sent to the CC.

When the SC performs an unsolicited refresh, the CC computes a new refresh rate as follows. The SC piggybacks the *write-sliding-window* on the unsolicited refresh. The CC uses these times to compute the request-numbers-window for the last k read-write requests. These numbers are taken as the λ_w and λ_r's, and the CC uses Formula 4 in order to compute the optimal refresh rate; this will become the new refresh rate, and the CC has to send a control message to the SC informing it of the new refresh rate. Thus, there is an extra cost incurred if the CC changes the refresh rate. Thus the CC will change the refresh rate only if the expected cost for the optimal refresh rate beats the cost of the current refresh rate by at least ω.

Before concluding this section, we make two remarks about the algorithm. First, it may be necessary to take into consideration that the CC and SC clocks are not synchronized. This may be a problem since the algorithm computes the request-numbers-window by selecting, from the read and write sliding windows, the *latest* k requests. In order for this set to be correctly computed, one has to account for the clocks' divergence. But this can be done easily if, once every g refreshes, the SC piggybacks the current value of its clock on the refresh message. Then the CC can compute the clocks' divergence, and either synchronize its clock to the SC's clock, or adjust the times in the sliding window to account for the divergence.

The second remark concerns k, the size of the sliding window. There are two opposite forces to be considered in choosing the value of k. If the values of the λ's change over time, then choosing a smaller k has the effect of discarding past λ's in favor of recent ones, i.e. adjusting to the current λ's. On the other hand, assume that each type of request is Poisson distributed with a fixed parameter. Then, by the law of large numbers, the larger the value of k, the closer are the values in the request-numbers-window to the actual λ's. However, in [6] we have shown that, even if the distribution parameters are fixed (but unknown), for a window of size 23, the expected cost of a sliding window algorithm comes within 4% of the optimum expected cost.

Thus we feel that a window of approximately 23 is appropriate, and using it the DDC algorithm will perform well even if the distribution parameters are fixed.

5 Worst case

In this section we analyze the worst case behavior of the Dynamic Divergence Caching algorithm and the Static Divergence Caching algorithm. We prove that the DDC algorithm is superior to the SDC algorithm.

The appropriate measure of the worst-case behavior of on-line algorithms is its *competitiveness* [10]. An on-line algorithm is an algorithm that receives its input schedule one request at a time, and acts on each request before obtaining the next one. Roughly speaking, an on-line algorithm is competitive if its cost is at worst a constant times the cost of any off-line algorithm, for any sequence of requests.

What is wrong with the traditional worst-case behavior of an algorithm? The answer is that the worst case of an on-line algorithm occurs when the input is chosen by a potent adversary trying to make the algorithm perform poorly. Then it makes no sense to ask "For Algorithm X, what is the maximum cost per request of any schedule?" The reason is that the adversary can construct a schedule by always making the next request one the algorithm must pay for. For example, let the control-message cost $\omega = 0$. For every algorithm considered in this paper, there is some schedule of length ℓ on which that algorithm incurs cost ℓ, so by that "worst-case" measure, all the algorithms have the same complexity.

What we really want to know is "How does the cost incurred by Algorithm X compare to the minimum cost that any algorithm must incur?" and it is this question that competitiveness answers.

Now let us give a precise definition of competitiveness. For an algorithm A and a schedule of requests σ, let $C_A(\sigma)$ denote the cost of Algorithm A on schedule σ. Then, on-line algorithm A is c-competitive if there is a constant k such that: for *any* schedule σ and *any* algorithm B (including all off-line algorithms that receive the entire schedule σ in advance) we have

$$c_A(\sigma) \leq c \cdot C_B(\sigma) + k.$$

An algorithm is *competitive* if it is c-competitive for some constant c.

For our problem, if requests really are generated by Poisson processes with fixed intensities, then one should use the

algorithm given in Section 3 above, and competitiveness is an inappropriate criterion. However, if the requests are generated at random, in a way that makes the past requests irrelevant in predicting the future ones, static algorithms can perform very poorly.

In this section we will show that the DDC algorithm of Section 4 has a bounded competitive ratio. We begin by presenting the optimal off-line algorithm, that we call O, for an arbitrary schedule.

Let σ be an arbitrary schedule. We break σ into blocks of reads and writes. Formally let $B_1, B_2, \ldots, B_\ell$ be the division of σ into blocks such that each block is either all reads or all writes, and adjacent blocks contain different kinds of requests. Assume that at the beginning of σ the CC has a fresh version of the data object.

Off-line algorithm O marks read blocks as follows. The first read block that is marked is the first read block preceded by a greater total number of writes than the minimum tolerance of any read in the block. Thereafter, each time there is a read block containing a read with tolerance less than the total number of writes since the last write paid for, Algorithm O marks that read block. Algorithm O propagates to the CC the last write in every write block preceding a marked read block. All the other reads and writes are local, and consequently incur 0 cost. Thus, the cost of O is the total number of marked read blocks.

Lemma 1 *Algorithm O is the optimal off-line algorithm.*[2]

Theorem 2 *The Static Divergence Caching algorithm has an unbounded competitive ratio.*

On the other hand, the DDC algorithm performs well, in the sense of being competitive. To prove this, we first need to examine just how well the optimal off-line algorithm performs.

Lemma 2 *Between any two read blocks marked by Algorithm O, there are at most $2M$ blocks.*

Theorem 3 *The Dynamic Divergence Caching algorithm is competitive.*

6 Experimental results

We have performed many experiments, where we randomly generated schedules of requests, and compared the algorithms' performance on those schedules. In this section, we summarize the results. In both subsections we compare the Dynamic Divergence Caching algorithm (hereinafter DDC) to the the Static Divergence Caching algorithm (hereinafter SDC). The difference between the subsections lies in the selection of the input schedules. In the first subsection the input schedules have fixed distribution parameters (λ's), and in the second subsection they vary over time. For all our experiments, we fixed the maximum tolerance of any read, denoted M, at 20.

6.1 Fixed distribution parameters

We generated schedules of 1600 requests, for each of 84 different values of the 21 parameters λ_w and λ_{r_j} for $1 \leq j \leq 20$. For each schedule σ we used formula 4 to compute the optimal refresh rate, k_σ. Then we ran on σ the DDC algorithm with various window sizes, and the SDC(k_σ) algorithm.[3]

In Figure 1 we show a grand summary of the data — the average of all 84 runs, for the case where the control message cost is negligible, namely $\omega = 0$. In particular, we plot the ratio of the average (over all runs) cost of the DDC algorithm with window size k (hereinafter DDC(k)) to the average cost of the optimal static algorithm, as a function of k. Notice that the optimal static algorithm differs from schedule to schedule.

The solid line at 1 represents the average cost of the optimal static algorithm for each case.

The main result of these experiments is that the performance of $DDC(k)$ improves sharply as k increases to about 23, and for $k \geq 23$ the cost of DDC(k) is only 10–15% greater than the cost of the best static algorithm. In fact, this pattern was observed for almost every schedule of the experiment, although the graph in Figure 1 shows only data for all the runs averaged together.

Two other important quantities are *not* shown in the graph. One is the performance of the optimal off-line algorithm. The average cost of the optimal static algorithm was typically about 2.25 times the average cost of the optimal off-line algorithm. Recall that the off-line algorithm has the advantage of seeing all the requests in advance, something that is impossible in real life.

We also compared the optimal static algorithm to the better (on the particular run) of the static algorithms with refresh rates 1 and infinity. This corresponds to traditional methods that do not allow for divergence caching, but only for caching (refresh rate 1) or not caching (refresh rate infinity) [6]. The optimal static algorithm showed, on average, a factor of two improvement. This demonstrates the power of divergence caching.

Note that the factor of two improvement is an average over 84 different settings of the λs. For certain values of the λs a refresh rate of 1 or infinity was the optimal value (typically with writes being either a very high or very low

[2] Proofs are omitted because of space limitations.

[3] SDC(t) is the SDC algorithm with refresh rate t.

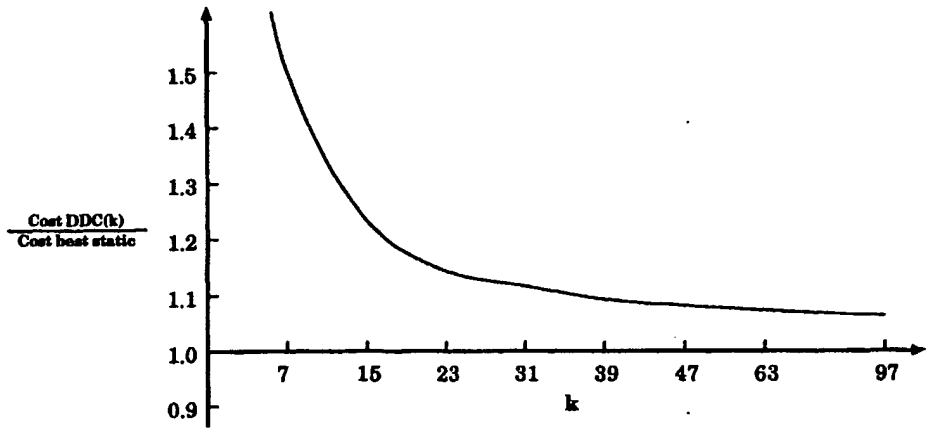


Figure 1: Experimental results for fixed λ's for $\omega = 0$ case. Graph shows ratio of costs paid by DDC(k) to costs paid by the optimal static algorithm.

percentage of all requests), and for certain values of the λs the improvement with divergence caching was much greater than fourfold.

We also computed the performance of our algorithms for values of ω ranging from 0.1 to 0.9. We performed the same experiments as for the case of $\omega = 0$. The results were broadly similar, with the following exception.

DDC(k)'s performance became worse as ω increased, especially for small values of k. For example, for the combination of experiments reported in Figure 1, with $\omega = 0$, DDC(23) paid 14.9% more than the optimal static divergence algorithm. However, for a large value of ω, it was clearly significantly better to use the best static algorithm than any dynamic algorithm. For $\omega = 0.9$, DDC(23) paid 44.1% more on average than the best static algorithm. We hypothesize that the high costs for the dynamic algorithms are caused by the cost of the control messages to reset the refresh rate.

6.2 Time-varying distribution parameters

As we describe above in Section 5, theory leads us to believe that if the λ's change over time, then the dynamic divergence caching algorithms will outperform all static algorithms. We ran several experiments where we varied the λs over time, and we summarize the results here. They confirm our expectation.

We begin with the case of $\omega = 0$. A typical experiment is presented in Figure 2, which reports the average of 40 runs of 1600 requests each. In each run, we picked each of the λ_{r_j}s uniformly at random from 1 to 100, and then uniformly at random picked the fraction p of all requests that would be writes. (Thus λ_w was set to be $p/(1-p)$ times the sum of the λ_r's.) Every 157 requests, the λ's were randomly assigned new values according to the same rules.

We determined empirically which one refresh rate gave the best performance of all the static algorithms over the forty runs. In figure 2 we plot the ratio of the average (over all 40 runs) cost of DDC(k) to the average cost of that static algorithm. The main result is that the cost of DDC(k) improves with k, with considerable improvement up to around $k = 23$, and slight improvement thereafter. For $k \geq 23$, the average cost of DDC(k) is roughly 70% of the cost of the best static algorithm.

The cost-advantage of the dynamic divergence caching algorithms varied with the method of changing the λs over time. If the ratio of writes to reads remained constant over time, while the individual λ_{r_j}s varied, then the dynamic divergence algorithms were only slightly better than the best static algorithm. On the other hand, much larger improvements than those shown in Figure 2 were found when we alternated periods where writes were at most 30% of all requests with periods where writes were at least 70% of all requests. In practice, alternating phases of read-intensive and write-intensive patterns might be a common.

Remember that these results are for the case $\omega = 0$. We ran the same experiments for values of ω ranging from 0.1 to 0.9. We once again found that the dynamic divergence algorithms DDC(k) are superior to the best static algorithm, again with a value of $k = 23$ seeming to be about where the gain levels off. However, the cost-advantage of DDC(k) decreased as ω increased. For example, for $\omega = 0.1$ the advantage of DDC(23) was 19.1% (versus an advantage of

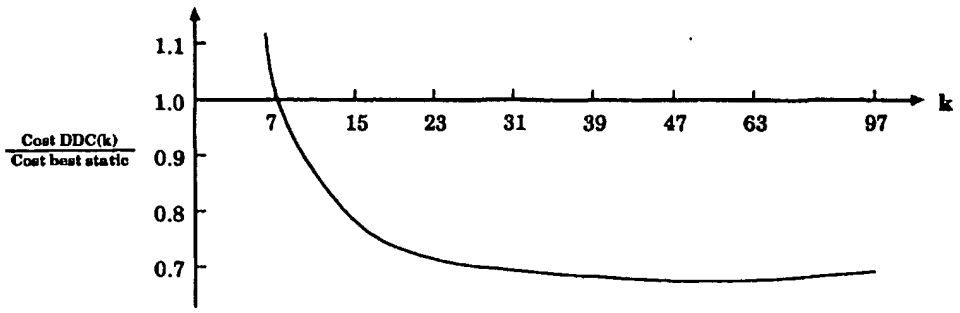


Figure 2: Experimental results for time-varying λ's for $\omega = 0$ case. Graph shows ratio of costs paid by DDC(k) to costs paid by the optimal static algorithm.

25.6% for $\omega = 0$). The cost-advantage decreased roughly linearly with ω to become near zero at $\omega = 0.9$. This is due to the increasing cost of control messages to reset the CC's refresh rate.

7 Relevant research

This paper is related to two topics, caching and replica divergence. We will first discuss related works on caching. Such works (e.g. [5, 11, 10, 13]) have assumed a consistent environment, i.e. one in which cached objects are copies of the most recent version. In our previous works (e.g. [6, 7, 8]) we have also made this assumption, and we have concentrated on a variant of caching called dynamic allocation. The objective of dynamic allocation is to minimize communication, i.e. the number of object transfers, rather than optimizing performance as in traditional caching. For example, given enough storage, traditional caching will keep a copy of every accessed object at the client computer; if the number of writes at the server computer is much higher than the number of reads at the client computer, then this approach will incur excessive communication. In contrast, dynamic allocation caches and uncaches an object depending on the read-write pattern.

Dynamic allocation in a weak-consistency environment, as studied in this paper, is a generalization of dynamic allocation in a strong consistency environment. Specifically, in a strong consistency environment a client computer is in one of two states. If the client computer has a copy of the data object, it means that the refresh-rate is 1 (i.e. every update is propagated to the client computer). If it does not have a copy of the object, then the refresh-rate is infinity. Dynamic allocation switches the refresh rate of a client computer between two values, 1 and ∞.

In a weak consistency environment these two states are extremes of a spectrum. Dynamic allocation permits intermediate values of the refresh-rate. Furthermore, the refresh-rate may vary dynamically depending on the read-write pattern. In other words, in a weak consistency environment the access pattern of an object determines not only the allocation scheme, but also the frequency at which the object is refreshed.

Now we will discuss related works on replica divergence. There are several studies of replica-divergence issues, such as [1, 2, 4, 12, 14]. These works assume, as we do, that each object has a computer that stores the most up to date version, and other computers store quasi-replicas that may diverge. The allowed divergence is specified by the user. In other words, a user manually specifies for each quasi-replica of an object a fixed divergence which results in a fixed refresh rate. For example, in [2], if the quasi-replica of object x at a particular computer c has a refresh rate of three, then every third update generates a refresh of x at c. Therefore, the divergence is specified at the quasi-replica level.

In contrast, in this paper we assume that the user specifies the allowed divergence, i.e. the tolerance, at the read level. The divergence of the quasi-replica is computed by an algorithm that we provide. It automatically varies the quasi-replica divergence, i.e. the refresh rate, in order to optimize costs. This optimization is not guaranteed by previous works. Moreover, the previous works do not discuss how to determine the refresh rate. The formulas that we obtain in this paper can be used for this purpose (assuming that the distributions of the reads with the various tolerances are known), and in this sense the present work is complementary to previous work.

8 Concluding remarks

In this paper we proposed a new mechanism, divergence caching, for use in client-server computing environments to

reduce the number of transmissions from database servers to clients. Divergence caching achieves this objective by giving a tolerance to each read of an object issued at a client computer, namely an allowed deviation from the most up-to-date version of the object; this is combined with a periodic refresh, namely a periodic transmission of the object from the on-line database to the client.

We proposed two algorithms based on divergence caching, Static and Dynamic. In static divergence caching the refresh rate is fixed, whereas in dynamic it varies over time. The first is appropriate when the access pattern to an object is fixed and known a priori, and the latter is appropriate in other cases.

These algorithms were analyzed in the worst case and in the expected case. We have shown that in the worst case the dynamic algorithm is strictly superior to the static one.

The results of the probabilistic analysis are as follows. We obtained a formula (4) for the optimal refresh rate of the static divergence caching algorithm, i.e. the refresh rate that minimizes the expected cost of the algorithm. The optimal refresh rate depends on the distributions of reads and writes of the object, and on the ratio of control-message-cost to data-message-cost.

Then we experimentally analyzed the dynamic algorithm using a large number of schedules that were probabilistically generated using Poisson processes. An important parameter of the dynamic algorithm is the size of the sliding window. We have shown that the appropriate window size is about 23.

We also showed that when the distributions of reads and writes are fixed, and a control-message-cost is relatively small compared to a data-message-cost, then the dynamic algorithm comes within 15% of the optimal static one; when the cost of the control message is higher the dynamic algorithm performs worse. The reason for this is that the dynamic algorithm sends additional control messages for the client computer to inform the server of refresh rate changes.

When the distributions of reads and writes vary over time, then the dynamic algorithm is superior to any static one, regardless of the ratio of control-message-cost and data-message-cost.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Quasi-copies: Efficient data sharing for information retrieval systems. In *Proc. of EDBT '88, LNCS 303*. Springer Verlag, 1988.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, 1990.

[3] B. R. Badrinath and T. Imielinski. Replication and mobility. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 9–12.

[4] D. Barbara and H. Garcia-Molina. The case for controlled inconsistency in replicated data. In *Proc. of the IEEE Workshop on replicated data*, 1990.

[5] M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita. Data caching tradeoffs in client-server DBMS architectures. In *Proc. 1991 ACM SIGMOD Internat. Conf. on Management of Data*, pages 357–366, 1991.

[6] Y. Huang, P. Sistla, and O. Wolfson. Data replication for mobile computers. In *Proc. 1994 ACM SIGMOD Internat. Conf. on Management of Data*, pages 13–24.

[7] Y. Huang and O. Wolfson. A competitive dynamic data replication algorithm. In *Proc. Ninth Internat. Conf. on Data Engineering*, pages 310–317, Vienna, Austria, 1993. IEEE.

[8] Y. Huang and O. Wolfson. Dynamic allocation in distributed systems and mobile computers. In *Proc. Tenth Internat. Conf. on Data Engineering*, pages 20–29, Houston, Texas, 1994. IEEE.

[9] T. Imielinski and B. R. Badrinath. Querying in highly mobile distributed environments. In *Proc. of the 18th International Conference on VLDB*, pages 41–52, 1992.

[10] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[12] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, pages 53–60, 1986.

[13] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Softw. Eng.*, 39(4):447–459, Apr. 1990.

[14] D. B. Terry. Caching hints in distributed systems. *IEEE Trans. Softw. Eng.*, pages 48–54, Jan. 1987.

Persistent Object Synchronization with Active Relational Databases

Fábio A. M. Porto[1], Sérgio R. Carvalho[2], Maurício J. Vianna e Silva[3], Rubens N. Melo[4]
Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225, Rio de Janeiro, RJ, 22453-900, Brazil
{porto[1], sergio[2], rubens[4]}@inf.puc-rio.br, mjvs@domain.com.br[3]

Abstract

One of the most common client/server architectures in enterprise systems today is the combination of object-oriented applications with active relational database systems. With this combination, developers have to overcome a difficult problem: the impedance mismatch between object orientation and the relational model. To date, there are several incomplete approaches for describing the integration of static and dynamic object aspects and active relational databases. An important issue missing from these approaches is the state synchronization between server tuples and client-cached objects. In a previous paper we proposed a technique for mapping the dynamic behavior of objects into active relational databases, using database triggers and stored-procedures. This paper extends our previous one with an architecture based on a replication strategy that maintains server tuples and client-cached objects synchronized with respect to state. This architecture automatically updates client-cached object versions when their corresponding server database tuples are updated.

Keywords: object orientation, active relational databases, state synchronization, data replication, object cache, stored procedures

1 Introduction

For more than a decade, business critical applications have been representing their information assets in relational database management systems (RDBMSs) [Duhl96]. More recently, advances in network technology, distributed programming and software architectures propose the organization of business applications in different tiers spread through various computers [Buschman+96, Aarsten+96, Hirschfeld96].

The combination of object-oriented applications with relational database systems within a client-server architecture is probably one of the most common choices for enterprise systems today [Delis+98]. Unfortunately, this combination has still many issues to overcome: the impedance between the Object Oriented (OO) model and the Relational model (RM) requires different approaches to deal with structural and behavioral clashes [Keller+96].

On the structural side, for example, object attributes may be stored in different database tables; also, object relationships such as inheritance have no counterpart in the relational world. On the

behavioral side, which we address in this paper, state changes in application objects must be reflected on their persistent versions, and vice-versa.

Pattern languages have been proposed to bridge the existing gap between the two technologies [Brown+96a, Keller+96, Silva+97]. In [Porto+98], we discussed this issue with respect to object behavior. Our proposal allows the representation of object life cycles within active RDBMSs, implementing object behavior via database triggers and stored procedures. By encapsulating dynamic behavior into the RDBMS we simplified the client side of the application and reused all rule enforcement mechanisms provided by commercial database systems.

However, the action part of server rules can, in many cases, change the state of stored objects. The problem of representing these changes in the client side presents itself. The objective of this work, which extends the previous one, is to propose an architecture for the automatic update of client object versions from alterations performed on the database version of the object. We therefore continue to address behavioral OO/RM clashes, neglecting to consider structural mismatches.

In this work, client-server applications with persistent objects stored into RDBMSs can be perceived as instances of an environment with replicated object versions. Once a set of persistent objects is read into the client station, at least two versions of the same object exist: a persistent version, stored into the RDBMS, and an application version to be accessed by the user in the client machine.

The remainder of this paper is organized as follows. Section 2 describes the object synchronization problem in more detail. Section 3 presents the proposed object synchronization architecture. Section 4 shows the scenarios where data are updated and synchronized. Section 5 exemplifies the use of the architecture with a simple application scenario. Section 6 compares our solution with related work. Section 7 presents conclusions and future research.

2 The Synchronization Problem

Consider an application implemented with an OO programming language and an active RDBMS. The application runs on a PC type desktop and the RDBMS on a server machine. Clients start the application and request data from the RDBMS, thus loading the application business objects. Once data gets loaded into the client application, the user deals with it as an independent OO application. The relational implementation becomes entirely transparent [Keller+96].

Application objects become versions of the persistent relational data, cached at the client machine, or at the application layer environment. The reasons for creating such data versions are initially to offer OO semantics for the application data stored in a RDBMS, and secondly to increase overall performance by splitting the application between various collaborating environments.

Considering updating applications, object versions in each client and within a (possibly distributed) server can be modified by user actions in some client's environment, and by active behavior in the RDBMS, via pre-defined triggers and stored procedures [Widom+96, Porto+98]. Thus, in the presence of either application- side or database-side updates, object versions cached in client environments become out of date, so-called stale cached data [Franklin+97].

This happens for three main reasons:

In order to increase overall throughput, persistent application classes implement an optimistic concurrency control protocol [Gray+93], so data read from the RDBMS is not locked while being processed by the client;

User updates take place offline, over local data versions. This strategy greatly increases client execution performance and provides an improvement for overall RDBMS data concurrent access. Updates go to the RDBMSs only at the end of the transaction, during the commit processing;

Server logic, implemented as stored procedures and triggers, may update persistent data that might have been cached and asynchronously updated within the client environment.

With increasing numbers of application clients and application update rates, the lack of synchronization within object versions may turn out to be critical. Client data lag far behind their persistent versions. In such a scenario, a great number of transactions may have to be completely re-submitted.

Considering the above, update-intensive applications can benefit from a proposal that addresses the state synchronization between client-side and database-side objects, thus aiding in the resolution of the OO/RM impedance mismatch. In the next section we present such an architecture.

3 The Object Synchronization Architecture

Figure 1 shows the classes and relationships we propose. The ConcreteApplication, ConcretePersistentApplication and Transaction classes are structured in ways very similar to patterns proposed in [Keller+96, Keller98, Silva+97]. The service classes ApplicationLog, CopyManagerClient and CopyManagerServer provide our proposed functionality.

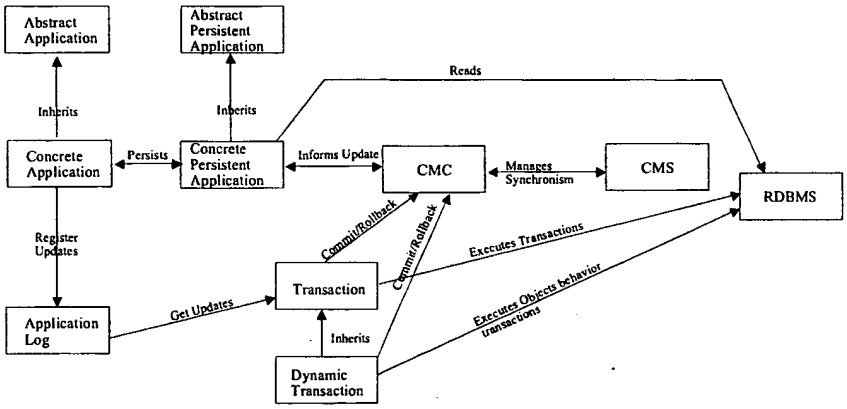


Figure 1 Object Synchronization Architecture Class Diagram

In the next paragraphs we shortly describe the participants in this architecture, their main responsibilities and collaborations. Section 4 presents the proposed synchronized behavior.

The AbstractApplication (AA) class generalizes the manipulation of application objects with persistent behavior. Each domain specific class modeling objects with persistent behavior extends this AA class. These ConcreteApplication (CA) classes model objects in application domains. Persistent read and write operations over CA class objects are passed to the corresponding ConcretePersistentApplication object, described below.

The AbstractPersistentApplication (APA) class provides a common persistent behavior for domain specific classes, including the special messages implementing the synchronization mechanisms between client objects and database tables. ConcretePersistentApplication (CPA) classes extend APA, modeling the persistent behavior of domain objects.

Each CPA is a Singleton [Gamma+95] responsible for the mappings between corresponding object views and relational tables. All communications involving object versions and RDBMS tables pass through one such CPA object, which acts as a broker. The kind of structural

mappings[1] that may be executed by a CPA class are those presented in [Keller+96]. They represent the main OO structural constructs, such as class, inheritance hierarchy, and association.

The Transaction class singleton controls client application transactions. The responsibilities of this singleton are: to generate transactions identifiers, to register new client transactions with the CMC object, to inform the CMC object of the success or failure of the corresponding transactions, and to apply updates over RDBMS data when a commit operation successfully concludes a client transaction.

The DynamicTransaction (DT) class extends the Transaction class with special behavior needed to process client transactions that execute stored-procedures, which in this work implement object behavior in the RDBMS. The DT object queries and updates data in the auxiliary tables used to inform the CMS object of updates executed by server procedures over persistent data.

The ApplicationLog (AL) class also models a Singleton. It registers the update operations executed by CA objects using a write-ahead policy [Gray+93]. Its object collaborates with the Transaction object during the commit processing, providing all the operations executed within the transaction's boundaries.

The CopyManagerClient (CMC) class models a Singleton responsible for the communication among all CPA objects and the Transaction object with the CopyManagerServer object. It provides communication transparency between clients and the CopyManagerServer object. All operations over cached data will be asynchronously informed by the CMC object in each client environment to the CopyManagerServer object, during transaction execution.

The CopyManagerServer (CMS) class models a Singleton multi-threaded server in a 3-tier environment, serving all RDBMS clients. It registers transactions running on client machines together with a list of table names corresponding to the objects loaded by the transactions. This data structure is used by the CMS as a directory for sending synchronization messages, which inform registered clients with cached versions of persistent objects, of updates committed by the RDBMS server procedures or by concurrent client applications.

We may consider the architecture as split in three tiers. Composing the client environment we have the following classes: AA, CA, APA, CPA, AL, Transaction, DynamicTransaction and CMC. In the middle tier runs the CMS server object and the RDBMS composes the third tier.

In the next section we describe a few scenarios illustrating the synchronized behavior we obtain with the architecture above.

4 Update Scenarios

The architecture presented in section 3 aims to reduce the time lag between a confirmed database update and the moment in which cached versions of the corresponding data have their attribute values synchronized. The database modifications are applied either by applications running on client machines and executing SQL statements through database connections, or by server procedures running on the database server machine.

This section integrates our synchronization architecture with the scenarios in which database data are modified. Our main concern is to present the scenario where server procedures update object data stored in the RDBMS. This is the case when we implement object behavior through database stored-procedures and triggers. We also discuss the modifications in database data executed by client applications. This scenario is divided in two parts: the process of loading and

---

[1] It is important to note that not all object views might be translated into tables. In special, non updateable views [Silberschatz+96, Elmasri+94] like the ones containing aggregated values do not map into tables modeled over the analytical data.

updating objects and the commit process. These scenarios are complemented with a fourth one that identifies and communicates registered clients of stale data.

In all four scenarios the main participants are the CMC and the CMS objects. They provide data structures and operations for supporting the synchronization process. We initiate the presentation of the update scenarios describing the responsibilities of these components.

The CMS object records, for each transaction in a client, a list of tables that have been accessed, and most importantly, have been updated during transaction execution.

The application successfully terminates a transaction by issuing the commit operation on the Transaction object. This event causes the staleness of cached versions being manipulated by other client environments. In order to synchronize data, the Transaction object informs the CMC object of the transaction's commit. The CMC object then passes the information to the CMS object through the InformCommit message. It's a function of the CMS object to inform registered clients that their data versions are out-of-date.

Once the CMC object, controlling a registered client transaction, receives a message about the existence of new versions of stored data, it notifies interested ConcretePersistentApplication objects. When all the clients have acknowledged the message, the CMS object destroys the object corresponding to the finished transaction.

4.1 Client side updates

Figure 2 and Figure 3 show scenarios where a ConcreteApplication object is updated. Object modifications are imposed by transaction operations. Transactions are initiated by calling the BeginTransaction( ) operation on the Transaction object. Each update operation over persistent ConcreteApplication objects informs the transaction object controlling its progress.

After having been associated to a transaction object, the ConcreteApplication object uses it in its communication with the CMC object. All information interchange references the transaction object, so that in the event of a commit, it becomes possible to identify which tables had their states changed during the transaction's processing.

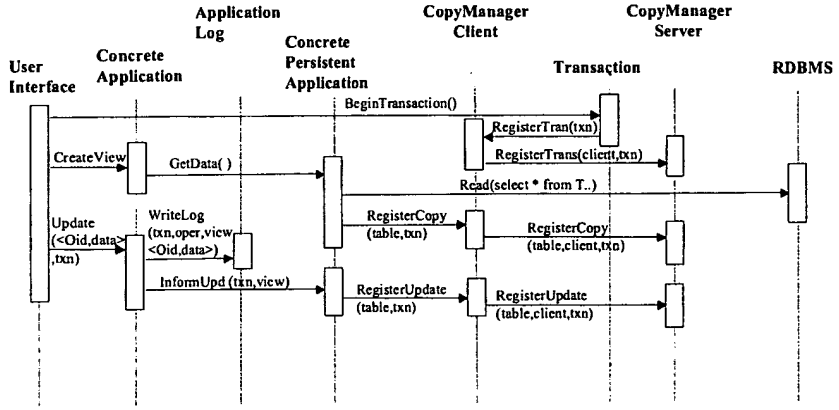


Figure 2 Object Synchronization Architecture Scenario – load and update Objects

Execution on the client side progresses with almost complete independence from the RDBMS side. Once an object is required, i.e. via an user interface request, the CPA object requests the proper tuples from the RDBMS and composes the object's view corresponding to the required ConcreteApplication object, for instantiation purposes. ConcreteApplication objects have a timestamp (ts) attribute. During instantiation, they receive the value of the oldest timestamp among its component tuples. This attribute will be used, during the RDBMS transaction commit, to validate the consistency of the cached version versus the RDBMS version.

An update in a ConcreteApplication object starts the synchronization process. This object informs its corresponding CPA object that an object has been updated. Considering the OO/RM mapping, the CPA object identifies the corresponding tables structurally associated with the updated object. Next, it informs the CMC object of the tables updated. Note that, if the object is composed by tuples in different relational tables, the CMC objects registers the update in all such tables. When the user decides to commit the transaction, it invokes the corresponding Transaction method. The commit process begins by identifying the update operations executed during the transaction, obtained by demanding the ApplicationLog object to provide the net effect of the operations executed during the period [Widom+96]. The CPA translates operations executed over objects into relational counterparts over tables. Once the set of operations is formulated, the Transaction object attempts to execute them within a single RDBMS transaction. During database updates, the corresponding client objects are locked, guaranteeing a consistent synchronization of views.

The Transaction object waits for the RDBMS's return from the commit operation. Following a successful return, it informs the CMC object of the transaction's commit. This object in turn informs participant clients of new database versions of data, of which they have stale versions. This is done by first informing the CMS object of the committed transaction, via the InformCommit message, with parameters identifying the client and the transaction that committed. The CMS object then invokes all clients registered for the updated objects.

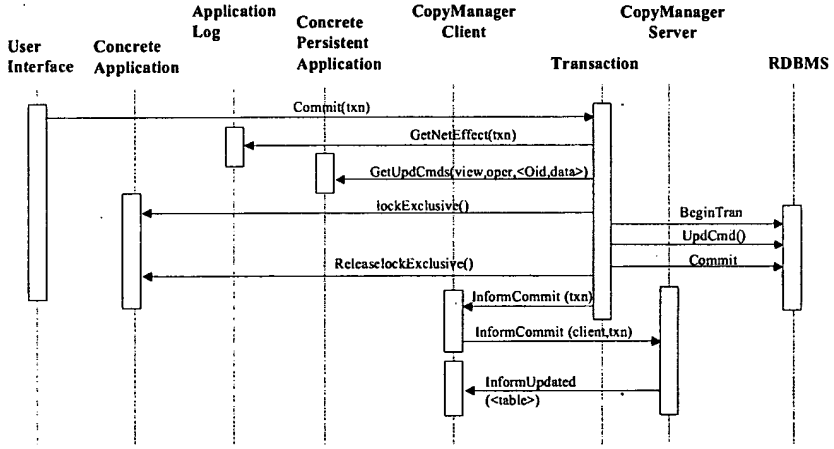


Figure 3 Object Synchronization Architecture Scenario - client transaction commit

4.2 Database side updates

Data may also be changed by RDBMS's server procedures. In particular, we propose the representation of client object behavior via triggers and stored procedures [Porto+98]. These server procedures may update, as part of their code, data cached in client environments.

Server procedures execute within the boundaries of a RDBMS transaction, controlled by the CPA object providing client object persistence. The CPA object uses a transaction object modeled by DynamicTransaction (DT) class to control transactions executing stored-procedures. The DT class extends the Transaction class overriding the operations for the creation and finalization of transactions.

To inform clients with cached object versions of updates executed by server code, we use three auxiliary tables: PersistentTable, UserTransactionTable and ProcedureUpdateTable. The PersistentTable is a meta-data table storing the table names and ids for the ones that have their states changed by server procedures. It serves two basic purposes: document tables representing

objects with behavior stored in the RDBMS, and provide consistency for data stored in the ProcedureUpdate table.

The UserTransactionTable is updated by the DynamicTransaction object during begin (insert) and end (delete) of transactions implementing object behavior. Its data represent the collection of transaction operations, through the association of the database user identification and the client's transaction identification.

The ProcedureUpdateTable is updated by the server procedures implementing object behavior. Examples of possible modifications imposed by server procedures include deleting and inserting tuples in state tables, and the execution of pre- and post-conditions associated with a state transition. Each of these updated tables is registered in the auxiliary table together with the user-id and client transaction id.

Figure 4 presents the execution scenario for procedures and triggers implementing object behavior. The overridden methods of class DynamicTransaction are responsible for inserting and deleting user transaction information into the UserTransactionTable.

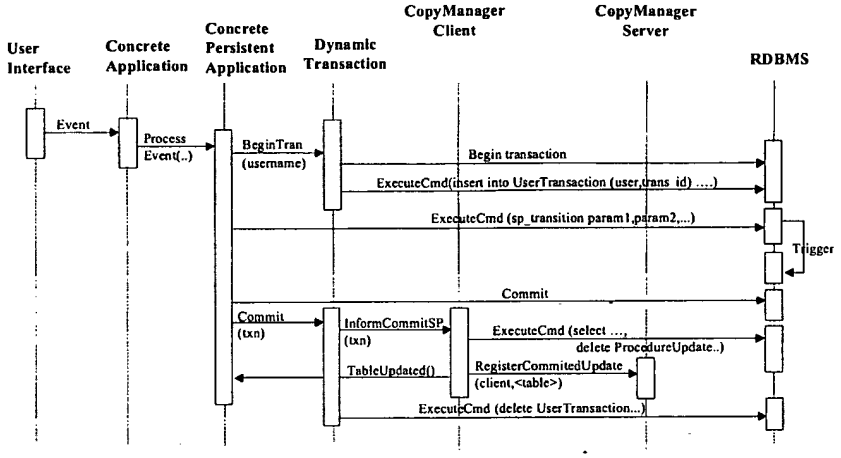


Figure 4 Server procedure's update scenario

After receiving a successful return code from the executed procedure, the CPA object commits the RDBMS transaction. It then terminates the client transaction by issuing the overridden DynamicTransaction Commit operation. The overridden operation controls the execution of a sequence of operations aiming to identify the tables updated during server processing and allowing for the initiation of the synchronization process. It also destroys the transaction object.

The CMC operation InformCommitSP, invoked by the Commit operation, executes in two steps: firstly it queries the ProcedureUpdateTable, finding out tables updated by server procedures, as shown by the query bellow.

```
"Select pt.table_name, timestamp From PersistentTables pt, ProcedureUpdate pu
    Where pt.table_id = pu.table_id and pu.user = :username and
    pu.trans_id =  :trans_id"
```

Secondly it deletes the corresponding tuples in the ProcedureUpdateTable, deleting the registration of updated tables. Having recovered the execution control, the Commit operation

deletes the tuple in the UserTransactionTable corresponding to the terminating transaction, deleting the transaction record.

Finally, it identifies the CPA objects associated with the updated tables and informs them of the RDBMS updates. To initiate the synchronization messages, the CMC object invokes the RegisterCommitedUpdate operation of the CMS object listing the tables that were updated during the execution of transactions.

4.3 Object level synchronization

The message sent by a CMC object to a CPA object informs that some data, corresponding to a view it controls, was changed. Considering that the granularity of the synchronization control exercised by CMC and CMS objects is a table, the CPA object is responsible for finding out if the change impacts some of the active objects under its scope. The CPA object invokes the GetObjects method of the corresponding ConcreteApplication object. The method returns a list containing the objects presently loaded at the client environment. Using its mapping rules, the CPA object queries the tables corresponding to its view. It uses the attributes composing the table primary key and the timestamp attribute value to identify the objects which need to be updated.

Basically, two sets of results are of interest. First, if no tuple is found for a primary key value, it means that the persistent version of the object has been deleted by some transaction. As a result, the version cached at the client must be destroyed. Second, if a database tuple exists for a primary key value but it presents a timestamp value greater then the one in cached object version, then the object persistent version has been updated. As a result, the object in cache must have its values updated.

Finally, with these queries the CPA object is able to identify the objects that had their states changed, lock them, update their versions and return a message to the user aborting the current transaction (see Figure 5).

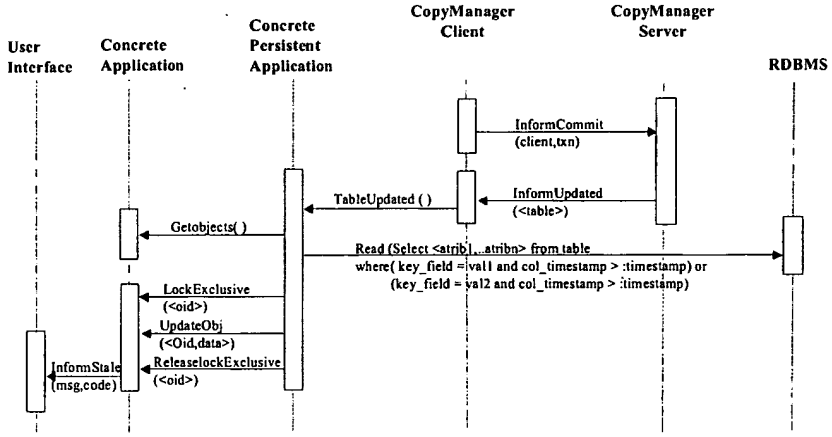


Figure 5 Object level synchronization

5 Related work

The problem of integrating OO applications with relational database systems has been the subject of various studies in the literature. [Keller+96] presents a relational access layer. We use this access layer as the basis for our work, and introduce a caching policy. Our proposal extends Keller's work with a treatment of the synchronization problem between client cached data and RDBMS server persistent data. The mapping rules implemented by the ConcretePersistenApplication class are those defined in [Keller+93]. In [Porto+98] we propose the modeling of object behavior through database stored-procedures and triggers. The patterns

presented in [Silva+97] extend those in [Keller+96]. In this work we use their StrongLayering pattern.

There is also a great deal of work associated with the replication strategy. For example, [Gonçalves+98] presents a pattern language for implementing architectures supporting object replication with different policies. It is a more general approach then that found in the Observer pattern [Gamma+95]. In our proposal, the CMS object is responsible for informing registered CMC clients of committed changes in the RDBMS. In a way, the CMC/CMS objects present a behavior similar to that found in the subject/observer metaphor.

Another area of related work investigates RDBMSs extended to manage client cached data. [Franklin+97] presents a taxonomy for algorithms that maintain the consistency of cached data. With respect to this work, our proposed concurrency control strategy relates to detection based protocols with validity checks deferred until the commit, and change notification hints sent after commit. We extend their approach dealing with object-cache consistency, taking into account object data that has been updated by stored procedures and triggers.

6 Conclusions and Future Work

One of the most common choices for software development these days combines three important paradigms: OO programming, active RDBMS and client-server architectures. The first two paradigms are not orthogonal, offering different modeling perspectives. The third paradigm often involves a distributed architecture on top of which one can distribute parts of the application. Putting all this together is not easy. Much work has been done in the OO/RM mapping. In this area one of the main concerns is the problem of synchronizing client-side object state changes and database-side table updates.

In a previous paper [Porto+98], we proposed the relational integration of application object behavior through database triggers and stored procedures. We used the active mechanisms of the RDBMS to execute object transitions, verify pre-conditions and execute pos-conditions. The execution of RDBMS server procedures, however, presents us the inverse problem: once the database code updates persistent data, the application objects become out of date.

In this paper we propose a solution to this synchronization problem, encompassing both the application-to-database solution described in [Porto+98] and a new solution to the inverse database-to-application problem.

The architecture we propose considers applications developed using an OO programming language. Persistent application objects are stored in relational database systems with the active capability of running server procedures. This architecture makes no further assumptions; the classes and relationships we propose can be implemented in any OO language, and the object-relational mappings may follow any proposed pattern language.

Persistent objects are created and processed in the client OO environment, as part of a client transaction running an optimistic concurrency control protocol. At the time a client transaction is confirmed, the application tries to store the object into the relational database.

Processing in the client environment is almost completely independent from the RDBMS. The data requested from the database are loaded in the client environment without being locked by the server. Once the client environments run independently from the server, updates made by server procedures may interrupt an ongoing client transaction. Depending on the size of the client's transaction, it may be very painful for the client to cancel all that has been done.

Our architecture aims to diminish the impact on client transactions by informing, as earlier as possible, of updates committed by other transactions that change the values of client objects. Our main concern is to inform clients of updates caused by server procedures and triggers during the processing of object state transitions.

Our architecture implements a combination of a replication strategy with client/server optimistic concurrency controls, and patterns for solving the OO/RM impedance mismatch. This

functionality may be summarized in three parts: the identification of updates over objects or tables, the registration of the updates and the broadcasting of the update message.

There are many opportunities for future work, examining alternative solutions to the problem. Our solution creates a RDBMS-like environment in the client. A possible alternative could be to use an OO DBMS in the client, simplifying part of the architecture. The control of updates by the CMS is done at table level; alternatively, we can control updates at the object granularity. Also, the communication between server RDBMSs and application components may be improved by eliminating the need to use auxiliary tables.

References

[Aarsten+96] A. Aarsten, D. Brugali, G. Menga, "Pattern for Three-Tier Client/Server Applications," In *Pattern Languages of Programs (PloP)*, Monticello, Illinois, 1996.

[Agarwal+95] S. Agarwal, C. Keene, and A. Keller, "Architecting Object Applications for High Performance with Relational Databases", In *OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX, October 1995.

[Brown+96a] K. Brown, and B. Whitenack, "Crossing Chasm: A Pattern Language for Object-RDBMS Integration", In J. Vlissides, J. Coplien, and N. Kerth (eds.), In *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, pp. 227-238.

[Brown96b] K. Brown, "Crossing Chasm: The Architectural Patterns," In *Pattern Languages of Programs (PloP)*, Monticello, Illinois, 1996.

[Buschmann+96] F. Buschmann, R. Meunier, P. Sommerlad, and M. Stal, "Pattern-Oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1996.

[Carey+91] M. Carey, M. Franklin, M. Livny, E. J. Shekita, "Data Caching tradeoffs in client-server DBMS architectures", In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Minneapolis, MI, May, 1991, Pages 357-366.

[Delis+98] A. Delis, N. Roussopoulus, "Techniques for Update Handling in the Enhanced Client-Server DBMS', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 3, May/June 1988.

[Duhl96] J. Duhl, "Integrating Objects with Relational Data", In *Object-Magazine*, SIGS Publication, March 1996, pp. 89-90.

[Elmasri+94] R. Elmasri, S. Navathe, "Fundamentals of Database Systems", Addison-Wesley, second ed., 1994

[Franklin+97] M. Franklin, M. Carey, M. Livny, "Transactional Client-Server Cache Consistency: Alternatives and Performance", In *ACM Transactions on Database Systems*, vol.22, No.3, September 1997, Pages 315-363.

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[Gonçalves+98] T. Gonçalves, A Rito, "Passive Replicator: A Design Pattern for Object Replication", In *EuroPLoP 97*, Munich, Germany, July 1997.

[Gray+93] J. Gray; A. Reuter , "Transaction Processing: Concepts and Techniques", Morgan Kauffmann, 1993.

[Hanson+98] E. Hanson, I. Chen, R. Dastur, K. Engel, V. Ramaswamy, W. Tan, C. Xu, "A flexible and recoverable client/server database event notification system", In *The VLDB Journal*, 7:12-24, 1998.

[Hirschfeld96] R. Hirschfeld, "Three-Tier Distribution Architecture", In *Pattern Languages of Programs (PloP)*, Monticello, Illinois, 1996.

[Keating+95] G. Keating and J. Thomas, "The Winning Combination: Object/Relational Solutions", In *Object-Magazine*, SIGS Publication, September 1995, pp. 64-67.

[Keller+93] A. Keller, R. Jensen, and S. Agarwal, "Persistence Software: Bridging Object-Oriented Programming and Relational Databases", In *ACM SIGMOD*, May 1993.

[Keller+96] W. Keller and J. Coldewey, "Relational Database Access Layer", In *Pattern Languages of Programs (PloP)*, Monticello, Illinois, 1996.

[Keller98] W. Keller, "Object/Relational Access Layers – A Roadmap, Missing Links and More Patterns", In *Europlop98*, Bad Irsee, Germany, 1998.

[Porto+98] F. A. M. Porto, M. Vianna e Silva, S. R. Carvalho, "Object Life-Cycles in Active Relational Databases", In *TOOLS-USA*, Santa Barbara, CA, August 1998.

[Silva+97] M. Vianna e Silva, S. R. Carvalho, J. Kapson, "Patterns for Layered Object Oriented Applications", In *Proceedings Second European Conference on Pattern Languages of Program Design (EuroPLoP 97)*, Munich, Germany, July 1997, pp. 85-94.

[Silberschatz+96] A. Silberschatz, H. Korth, S. Sudarshan, "Database System Concepts', M^c Graw-Hill, third ed., 1996.

[Yourdon+95] E. Yourdon, K. Whitehead, J. Thomann, K. Oppel, P. Nevermann, "Mainstream Objects: An Analysis and Design Approach for Business", Prentice Hall, 1995.

[Widom+96] J. Widom, S. Ceri, "Active Database Systems: Triggers and Rules for Advanced Database", Addison Wesley, 1996.